Science & Technology Facilities Council

Technical Report

A DAG-based parallel Cholesky factorization for multicore systems

J. D. Hogg

October 2008. Revised December 2009.

RAL-TR-2008-029

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

ISSN 1358-6254

RAL-TR-2008-029

A DAG-based parallel Cholesky factorization for multicore systems

J. D. Hogg[1]

ABSTRACT

Modern processors have multiple cores, making multiprocessing essential for competitive desktop linear algebra. Asynchronous processing with much inherent parallelism can be derived by using a directed acyclic graph (DAG) to represent the data dependencies between tasks.

In this paper, we present our implementation of a DAG-based Cholesky factorization, comparing several different scheduling approaches for the prioritisation of tasks. We demonstrate that complex scheduling approaches offer only a small performance improvement over very simple heuristics.

Our factorization is implemented in Fortran 95 using OpenMP.

Keywords: Cholesky factorization, DAG-based, symmetric linear systems, parallel, Fortran 95, OpenMP.

AMS(MOS) subject classifications:

---

[1] Computational Science and Engineering Department, Rutherford Appleton Laboratory, Chilton, Oxfordshire, OX11 0QX, England, UK.
Email: jonathan.hogg@stfc.ac.uk
Work supported by EPSRC grant EP/F006535/1.
Current reports available from `http://www.numerical.rl.ac.uk/reports/reports.html`.

December 24, 2009

1 Introduction

A common task in scientific software packages is solving linear systems

$$A\boldsymbol{x} = \boldsymbol{b}$$

where A is a (possibly large) dense positive definite symmetric matrix of order n and $\boldsymbol{b}$ is the known right hand side. We follow a standard approach of obtaining the Cholesky factorization

$$A = LL^T$$

and performing forward and backward substitutions, solving

$$\begin{aligned} L\boldsymbol{y} &= \boldsymbol{b} \\ L^T\boldsymbol{x} &= \boldsymbol{y}. \end{aligned}$$

Serial techniques for the solution of such problems on cache-based architectures are well known, for example LAPACK [2] and `HSL_MA54` [1] (the latter also supports partial factorizations, see Section 5). ScaLAPACK [3] and others present both shared and distributed memory variants of these algorithms utilising fork-join parallelism. Recent papers from Buttari et al. [4, 5] indicate that approaches based on directed acyclic graphs (DAGs) result in much better parallel speedup because of relaxed synchronisation constraints. Related work focusing more on BLAS operations is considered within the FLAME architecture [7].

In this paper, we present our implementation of a Cholesky factorization based on a task DAG approach, proposing new prioritisation schemes and performing a comparison with others. Section 2 describes the task DAG of a Cholesky factorization, while Section 3 shows how we execute such a DAG given some prioritisation. In Section 4 we present our approach to prioritisation. Details of our implementation are given in Section 5. Finally, in Sections 6 and 7, we give numerical results and our conclusions.

2 Cholesky task DAGs

A standard right-looking block factorization of a matrix, such as that shown in Algorithm 1, divides the matrix A into blocks A_{ij} with a given block width nb — giving us $nblk$ blocks in each row or column (the last block of a row or column may be smaller than nb). The steps are largely analogous to those of a standard Cholesky factorization.

We denote the block operations by the notation factor(j), solve(i, j) and update(i, j, k) to get the form shown in Algorithm 2. We can reorder many of these operations, for example we can move to a left-looking factorization (such as Algorithm 3), or one of several other variants. In fact we can have any variant we like so long as we have the data we are using in each operation "ready". We capture these data dependencies using a DAG, where an arrow from task 1 to task 2 means that we cannot start task 2 until task 1 has completed. Figure 2.1 shows the task DAG for the 4×4 case. The specific dependencies are:

factor(i) requires update(j, k, j) for $k = 1 \ldots j-1$.

solve(i, j) where $i \geq j$ requires factor(j) and update(i, k, j) for $k = 1 \ldots j-1$.

update(i, j, k) where $i, k > j$ and $i \geq k$ requires solve(i, j) and solve(k, j).

We can execute many of these operations in parallel, we merely need to obey the task DAG to ensure that we have the necessary data.

In the remainder of this paper, we shall refer to the task DAG using standard graph terminology. Each task is represented by a node, with dependencies represented by directed edges. If an edge from node i to node j exists then j is a child of i, with i as a parent of j. A node with no parents is a root and a node with no children is a leaf. A path is an ordered sequence of nodes connected by edges. Node j is a descendant of node i if there exists a path from node i to node j.

Algorithm 1 Standard right-looking block factorization, overwriting A with L

for $j = 1, nblk$ **do**
  $L_{jj} \leftarrow \text{factor}(A_{jj})$
  for $i = j+1, nblk$ **do**
    $L_{ij} \leftarrow A_{ij} L_{jj}^{-T}$
    for $k = j+1, i$ **do**
      $A_{ik} \leftarrow A_{ik} - L_{ij} L_{kj}^{T}$
    end for
  end for
end for

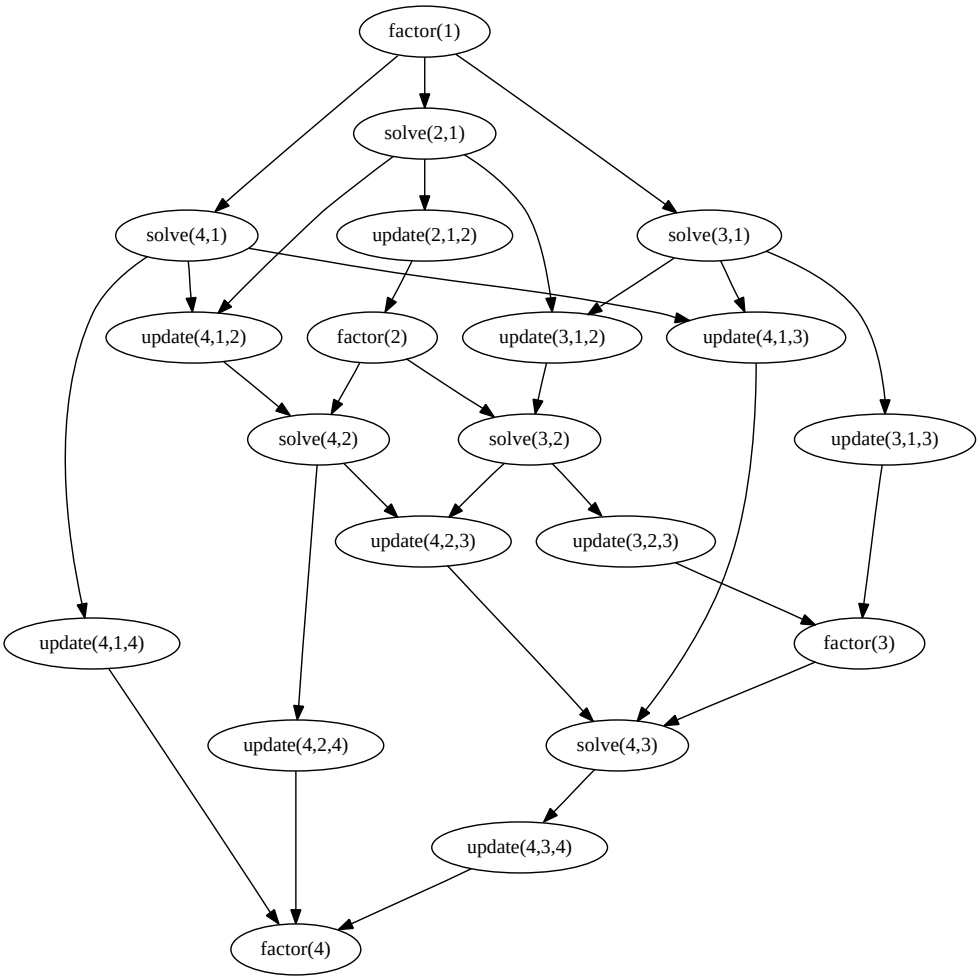
Algorithm 2 Right-looking block factorization

for $j = 1, nblk$ **do**
  factor(j)
  for $i = j+1, nblk$ **do**
    solve(i, j)
    for $k = j+1, i$ **do**
      update(i, j, k)
    end for
  end for
end for

Algorithm 3 Left-looking block factorization

for $j = 1, nblk$ **do**
  for $k = 1, j-1$ **do**
    for $i = j, nblk$ **do**
      update(i, k, j)
    end for
  end for
  factor(j)
  for $i = j+1, nblk$ **do**
    solve(i, j)
  end for
end for

Figure 2.1: Task DAG for a 4×4 block Cholesky factorization



3 Task dispatch

Algorithm 4 shows pseudo-code for how we might execute a task DAG using a task heap. The task heap contains tasks, with subroutines get_task() and add_task() that allow us to retrieve a task or add a new one. The algorithm retrieves a task, does the necessary work and then updates dependency information of other tasks, adding any for which the dependencies have been satisfied.

Algorithm 4 Task dispatch code for a simple execution of the task DAG

```
Initialise dep(i,j) for 1 ≤ i ≤ j ≤ nblk to number of parents of corresponding factor or solve nodes
loop
  call get_task(task)
  select case(task)
  case(factor)
    Perform Cholesky factorization of block (j,j)
    for i = j+1, nblk do
      dep(i,j) = dep(i,j) - 1
      if dep(i,j) == 0 then call add_task(solve(i,j))
    end for
  case(solve)
    L_ij = A_ij L_jj^{-T}
    dep(i,j) = -2        ! Flag that this solve has been performed
    for k = j+1, i do
      if dep(k,j) == -2 then call add_task(update(i,j,k))
    end for
    for k = i+1, nblk do
      if dep(k,j) == -2 then call add_task(update(k,j,i))
    end for
  case(update)
    A_ik = A_ik - L_ij L_kj^T
    dep(i,k) = dep(i,k) - 1
    if dep(i,k) == 0 then
      if i ≠ k then
        call add_task(solve(i,k))
      else
        call add_task(factor(k))
      end if
    end if
  end select
end loop
```

The dep(:,:) array is initialised before we start to contain the number of tasks that must be executed for a block before we can execute a factor or solve task as appropriate. Each time a dependency (either an update from the left or the factorization of the diagonal block above) is met we decrement this count, adding the task when there are no more dependencies to satisfy (clearly the dependency count is equal to the number of edges entering the corresponding factor or solve node in the task DAG).

Once the task solve(i, j) has been placed in the task heap, the element dep(i, j) can be reused as a flag to indicate whether that particular solve task has completed. After completion of the task solve(i, j) we scan through the dep elements of column j checking for update tasks that are ready to be added to the task heap.

To run this algorithm in parallel (shared memory) we need to prevent data races. A data race occurs when two threads are trying to read or write to the same location at once. We avoid this in our code

through the use of locks, as shown in Algorithm 5. We require locks to prevent multiple threads accessing the same dep(:,:) variable simultaneously. While we could use an OpenMP critical section (a section of the code that only one thread can execute at a time), the use of specific locks allows many of these operations to take place in parallel. Following a solve we insist that only one thread looks at the entire column trying to add updates to avoid having the same task added more than once. We minimise the amount of time spent while holding a lock by performing expensive numerical operations before acquiring them. Of specific interest is the need to prevent more than one thread performing an update operation on the same block simultaneously. To this end we have a special set of locks (in the array update_lock(:, :)). If a thread is unable to acquire the lock for its target block, it performs the update into a buffer T, which is then added to a list of delayed updates for later application. Before performing the factor or solve task for the target block, all pending updates to the block from the list are performed. Note that there is a lock in the get_task() and add_task() subroutines that prevents more than one task modifying the task heap at once.

As the algorithm is described, we can clearly generate a large number of temporary buffers T. A number of implementational tricks may be employed to reduce the memory used:

- If we require an update buffer, attempt to add to an existing one for that block. This requires an additional lock to be associated with each update buffer. This technique limits the number of buffers for any given block to the number of processors, though we would expect far fewer.
- If we have exhausted available memory for update buffers then we can instead wait to acquire a lock on the block we need to update. This strategy could however result in performance issues if the memory limit is often encountered.

4 Prioritisation

We now consider associating with each task in the heap of available tasks a number that we will refer to as a schedule. Calls to get_task() will always return the task on the heap with the lowest schedule. This allows us to prioritise tasks so as to keep the number of available tasks on the heap large — thus ensuring that processors are rarely idle waiting for tasks.

Buttari et al. [5] recommend always ensuring that tasks on the critical path (which they unusually define as the path connecting all nodes with the highest number of outgoing edges) are executed with a high priority. We consider three variations of their scheduling strategy:

- All nodes of the same type have the same priority. Nodes that typically have a higher number of outgoing edges are scheduled first. (i.e. any factor nodes are scheduled first, then solves, then updates)
- Nodes with a higher number of outgoing edges are scheduled first.
- Nodes with a higher number of descendants are scheduled first.

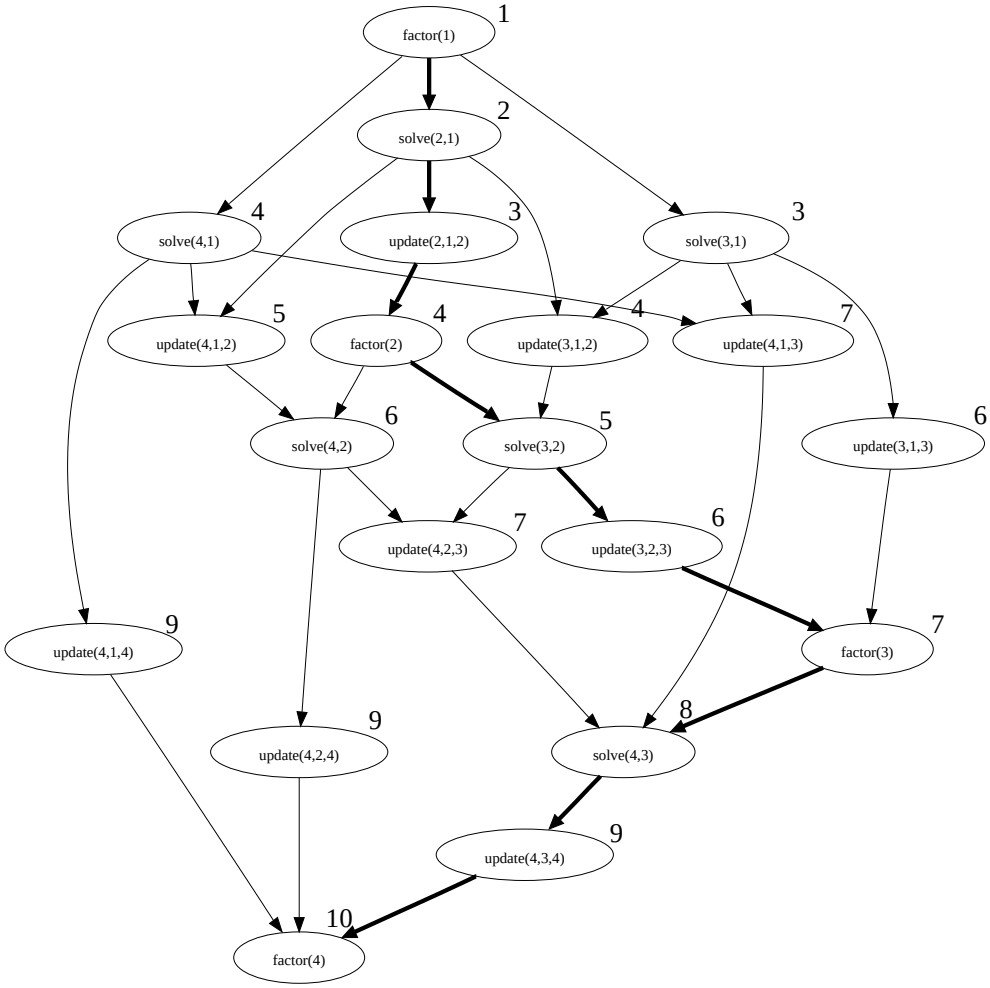
The first two options are easily realised; for the third option we can calculate the number of descendants using the following recurrence relations

$$
\begin{aligned}
\text{descendants}(\text{factor}(j)) &= \#\text{ solves in col } j + \#\text{ updates from col } j + \text{descendants}(\text{factor}(j+1)) + 1 \\
&= (nblk - j) + \tfrac{1}{2}(nblk - j)(nblk - j + 1) + \text{descendants}(\text{factor}(j+1)) + 1 \\
\text{descendants}(\text{solve}(i, j)) &= \#\text{ updates from block} + \text{descendants}(\text{solve}(i, j+1)) + 1 \\
&= (nblk - j) + \text{descendants}(\text{solve}(i, j+1)) + 1 \\
\text{descendants}(\text{update}(i, j, k)) &= \begin{cases} \text{descendants}(\text{factor}(k)) + 1 & i = k \\ \text{descendants}(\text{solve}(i, k)) + 1 & i \neq k \end{cases}
\end{aligned}
$$

Algorithm 5 Task dispatch code with synchronisations

```
loop
  call get_task(task)
  select case(task)
  case(factor)
    Apply any updates from update list
    Perform Cholesky factorization of block (j, j)
    for j = i + 1, nblk do
      Acquire lock(i, j)
      dep(i, j) = dep(i, j) − 1
      if dep(i, j) == 0 then call add_task(solve(i, j))
      Release lock(i, j)
    end for
  case(solve)
    Apply any updates from update list
    L_ij = A_ij L_jj^{-T}
    dep(i, j) = −2
    Acquire lock(j, j)
    for k = j + 1, i do
      if dep(k, j) == −2 then call add_task(update(i, j, k))
    end for
    for k = i + 1, nblk do
      if dep(k, j) == −2 then call add_task(update(k, j, i))
    end for
    Release lock(j, j)
  case(update)
    if Can acquire update_lock(i, k) then
      A_ik = A_ik − L_ij L_kj^T
      Release update_lock(i, k)
    else
      T = −L_ij L_kj^T
      Place T upon update list for block A_ik.
    end if
    Acquire lock(i, k)
    dep(i, k) = dep(i, k) − 1
    if dep(i, k) == 0 then
      if i ≠ k then
        call add_task(solve(i, k))
      else
        call add_task(factor(k))
      end if
    end if
    Release lock(i, k)
  end select
end loop
```

Figure 4.2: Scheduling for a 4×4 Cholesky factorization



which have closed form solutions

$$\begin{aligned}
\text{descendants}(\text{factor}(j)) &= \tfrac{1}{6}(nblk-j)(nblk-j+1)(nblk-j+5)+(nblk-j) \\
\text{descendants}(\text{solve}(i,j)) &= i\left(nblk-\tfrac{1}{2}i+\tfrac{3}{2}\right)-j\left(nblk-\tfrac{1}{2}j+\tfrac{3}{2}\right)+\text{descendants}(\text{factor}(i)) \\
\text{descendants}(\text{update}(i,j,k)) &= \begin{cases} \text{descendants}(\text{factor}(k))+1 & i=k \\ \text{descendants}(\text{solve}(i,k))+1 & i\neq k \end{cases}
\end{aligned}$$

Each of these approaches runs the risk of not prioritising a long chain of tasks that could end up being left to the end and causing task starvation for some processors.

In this paper, we shall define the critical path as the longest path from a root node to a leaf node. This represents the shortest sequence of events that must be executed in serial if an infinite number of processors is available and all tasks take the same amount of time (though the latter of these assumptions is not generally the case and we will relax this assumption later). Figure 4.2 shows the critical path for our 4×4 example marked on the graph as the bold edges.

If we continue our assumptions that all tasks take an equal amount of time and that we have sufficient processors to complete the critical path scheduling on time, then we can consider our tasks to occupy discrete time slices, numbered from 1. Our schedule number is derived as the latest time slice in which a task can be scheduled so that we can still meet the critical path. The numbering on Figure 4.2 shows such a scheduling for our 4×4 example.

In general the critical path is unique and consists of all the factor tasks and the shortest path between them, that is, the tasks solve$(j, j-1)$ and update$(j, j-1, j)$ for $j = 1, \ldots, nblk$. We can thus derive the schedules for these tasks as

$$\begin{array}{lcll} \text{schedule}(\text{solve}(j, j-1)) & = & 3j-4 & j = 2, \ldots, nblk \\ \text{schedule}(\text{update}(j, j-1, j)) & = & 3j-3 & j = 2, \ldots, nblk \\ \text{schedule}(\text{factor}(j)) & = & 3j-2 & j = 1, \ldots, nblk \end{array}$$

We can now work with a recurrence for non-critical tasks

$$\begin{array}{lcl} \text{schedule}(\text{update}(i, j, k)) & = & \begin{cases} \text{schedule}(\text{solve}(i, k)) - 1 & i \neq j \\ \text{schedule}(\text{factor}(k)) - 1 & i = j \end{cases} \\ \text{schedule}(\text{solve}(i, j)) & = & \min\left(\min_{k=j+1,i} \left[\text{schedule}(\text{update}(i, j, k))\right], \min_{k=i+1,nblk} \left[\text{schedule}(\text{update}(k, j, i))\right] \right) - 1 \end{array}$$

These equations are satisfied by the following closed form solutions:

$$\begin{array}{lcl} \text{schedule}(\text{factor}(j)) & = & j + 2j - 2 \\ \text{schedule}(\text{solve}(i, j)) & = & i + 2j - 2 \\ \text{schedule}(\text{update}(i, j, k)) & = & i + 2k - 3 \end{array}$$

If we now consider the case where we have insufficient processors to meet the critical path (i.e., we cannot schedule tasks in such a way that the final task finishes in the time slot our analysis has given it), is this schedule giving us a good prioritisation? Not always — consider a large collection of tasks that do not lie on the critical path, but require the completion of a number of dependencies before they become available. The dependencies for these tasks are not prioritised and, as a result, few tasks are available early in the execution sequence. However, due to the regular nature of a dense Cholesky factorization, we believe that the above schedule gives a near optimal scheduling provided that a sufficiently small block size is used.

We also consider relaxing the assumption that all tasks take the same amount of time. We assume all blocks are of size nb and consider the flop counts to be as follows:

$$\begin{array}{lcl} \text{flops}(\text{factor}(j)) & = & \frac{1}{3}nb^3 + O(nb^2) \\ \text{flops}(\text{solve}(i, j)) & = & nb^3 \\ \text{flops}(\text{update}(i, j, k)) & = & \begin{cases} 2nb^3 & i \neq k \\ nb^3 & i = k \end{cases} \end{array}$$

We hence give the factor, solve and update tasks weights of $1, 3$ or 6 respectively and then ask again for a recurrence relation to find the schedules, this time with no assumption on the critical path.

$$\begin{array}{lcl} \text{schedule}(\text{factor}(j)) & = & \min_{i=j+1,nblk} (\text{schedule}(\text{solve}(i, j))) - 1 \\ \text{schedule}(\text{solve}(i, j)) & = & \min\left(\min_{k=j+1,i} \left[\text{schedule}(\text{update}(i, j, k))\right], \min_{k=i+1,nblk} \left[\text{schedule}(\text{update}(k, j, i))\right] \right) - 3 \\ \text{schedule}(\text{update}(i, j, k)) & = & \begin{cases} \text{schedule}(\text{solve}(i, k)) - 6 & i \neq j \\ \text{schedule}(\text{factor}(k)) - 3 & i = j \end{cases} \end{array}$$

With these new relations, we observe that there is no longer a single unique critical path. Instead, we identify the set of tasks belonging to any critical paths as factor(1), factor$(nblk)$, all solves and all updates of the form update$(i, j, j+1)$. This represents the sequence of updating the whole of the next column as soon as possible. The reason this is so different from the old critical path is that the process of updating and factoring the diagonal is faster than that of updating the first column we need to solve. All solves are on the critical path as the critical path needs to reach each column for all block rows simultaneously.

The following formulae represent an explicit solution of the above

$$\begin{aligned}
\text{schedule}(\text{factor}(j)) &= \begin{cases} 9(j-1)+1 & j \neq nb \\ 9(j-1)-1 & j = nb \end{cases} \\
\text{schedule}(\text{solve}(i,j)) &= 9(j-1) \\
\text{schedule}(\text{update}(i,j,k)) &= \begin{cases} 9(j-1)-5 & i = k \\ 9(j-1)-4 & i \neq k, k \neq j+1 \\ 9(j-1)-2 & i \neq k, k = j+1 \end{cases}
\end{aligned}$$

5 Implementation

We implemented the algorithm described in the previous sections as the code `HSL_MP54` in the `HSL` software library [6]. It is written in Fortran 95 and has been primarily designed to replace the current partial factorization kernel (`HSL_MA54`) used in that library which currently makes use of fork-join parallelism. We use the OpenMP 2.5 standard to implement our shared memory parallelism.

In addition to a standard (complete) Cholesky factorization, our code also supports a less common but related problem that arises as a dense subproblem in sparse multifrontal matrix factorizations. This is the partial factorization

$$A = \begin{pmatrix} \mathcal{A}_{11} & \mathcal{A}_{21}^T \\ \mathcal{A}_{21} & \mathcal{A}_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} I & \\ & S_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ & I \end{pmatrix}$$

where the columns of A have been partitioned into two sets, one of which is fully factorized and the other is updated by this factorization. The corresponding forward and backward substitutions are

$$\begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \boldsymbol{y} = \boldsymbol{b} \text{ and } \begin{pmatrix} L_{11}^T & L_{21}^T \\ & I \end{pmatrix} \boldsymbol{x} = \tilde{\boldsymbol{y}}.$$

Clearly these operations can be performed using a variant of the algorithm we have so far described, which just omits the relevant factor task when it reaches the second set of columns. As our scheduling still prioritise factorizing the diagonal elements and solving we do not see any need to rework these schedules for this case, and indeed the surfeit of update tasks will allow us to smooth the tail of the computation.

Our code uses a job queue paradigm. Each factorization or forward/back solve is considered as a job, which is itself composed of many tasks such as those described in the previous sections. We have subroutines that place jobs in the job queue, but do no work on it, and a work subroutine that actually performs these jobs. The current state of the job queue and associated task heap are stored in a shared variable of type `mp54_keep`. This allows multiple instances of `HSL_MP54` to be run using the same team of threads if multiple independent jobs are available. Each thread can join a pool of workers at any stage in a job, picking up the next available task when it joins. This ability allows easy load balancing to be achieved at a level above that of our code, for example in a tree-based parallel factorization of a sparse matrix.

We enforce the condition that for each job queue, no job may start before its predecessor has completed. This simplifies the coding of the prioritisation and prevents naive users from introducing data races.

Internally we use a blocked hybrid form similar to that of Anderson et al. [1] for storing the data in a BLAS-compatible fashion while still using minimal storage. The user's data is by default rearranged from a lower packed format to this format in parallel with the factorization. Any columns that are not pivoted upon in a partial factorization are rearranged back to lower packed format to enable the user to easily perform manipulations such as assembly in to other matrices.

Solves have also been written in a task DAG fashion but, using the block size from the factorization, these are not as efficient as using a smaller block size. This is because a solve produces $O(n^2)$ tasks compared to the $O(n^3)$ from a factorization. We expect the overheads of such a rearrangement to outweigh any gains unless a large number of solves are required, and so have not pursued this avenue of research.

Table 6.1: Specification of test machines

	2-way quad Harpertown (`fox`)	16-way Power5 (`HPCx`)
Architecture	Intel(R) Xeon(R) CPU E5420	IBM Power5
Clock	2.50Ghz	1.50 Ghz
Cores	2×4	16×1
Theoretical peak (1/8 cores)	10 / 80 Gflop/s	6 / 96 Gflop/s
DGEMM peak (1/all cores[1])	9.3 / 72.8 Gflop/s	5.0/70.4 Gflop/s
Memory	8GB	32GB
Compiler	Intel 10.1 with `-fast`	IBM xlf 10.1
BLAS	Intel MKL 10.0	IBM ESSL

[1] Measured by using MPI to run independent matrix-matrix multiplies on each core

Our code implements two simple performance enhancements over the Algorithm of Section 3. Firstly, a thread will keep a task for itself if it would otherwise add it to the stack with the highest priority. This improves cache efficiency as the task will reuse some of the data from the current task. Secondly, as discussed in Section 3, we limit the amount of space required for the delayed update by looking for an existing update to the same block; if we find one we apply the update to it.

We note that for small ($n < 2000$) matrices, tuning of our code proved difficult due to context switching of our codes by the host operating system. Further, because of overheads inherent in communication, our code runs slower in parallel than in serial for very small matrices ($n < 200$ on our test machine). We were able to increase the performance in these circumstances by careful tuning of the get_task() routine so only one thread is ever spinning while waiting for tasks to appear in the queue — the others wait on a lock, making use of the far more efficient spinlocking mechanism of the OpenMP implementation. This seems to have helped reduce the number of cache misses on the threads doing actual work caused by use of the flush directive on idle threads.

6 Numerical Results

We present results mainly on our test machine `fox` (detailed in Table 6.1), however we also supply limited results from runs on a single node of the UK supercomputer `HPCx`. For each set of parameters (code, scheduling variant, order of matrix `n`, and number of threads `nthread`) we experimented to find the optimal block size; the reported results are for these optimal choices.

6.1 Choice of scheduling technique

We consider results for the five different scheduling methods discussed in Section 4. We shall refer to them by the following names

Simple All tasks of the same type have the same priority (method of Buttari et al.)

Max Child Tasks are prioritised by their number of children. The node with the largest number of children goes first.

Max Descendants Tasks are prioritised by their number of descendants. The node with the largest number of descendants goes first.

Fixed-CP Tasks are scheduled according to the critical path assuming all tasks take the same amount of time.

Weighted-CP Tasks are scheduled according to the critical path with weightings based on their operation counts.

Table 6.2: Comparison of different scheduling for complete factorization of matrices on 8 threads. Speed is measured in Gflop/s on `fox`, and percentages in brackets show performance gain over the Simple strategy.

n	Simple	Max Child		Max Descendants		Fixed-CP		Weighted-CP	
100	1.6	1.6	(0%)	1.6	(0%)	1.6	(0%)	1.6	(0%)
200	6.1	6.1	(0%)	5.9	(-4%)	6.5	(7%)	6.1	(0%)
300	10.0	10.1	(1%)	10.1	(1%)	10.6	(6%)	10.6	(6%)
400	13.7	13.7	(0%)	14.3	(4%)	14.5	(6%)	14.1	(3%)
500	16.9	17.0	(1%)	17.4	(3%)	18.1	(7%)	17.7	(5%)
1000	28.6	28.6	(0%)	28.7	(0%)	29.7	(4%)	29.1	(2%)
1500	35.9	35.8	(0%)	35.3	(-2%)	35.8	(0%)	35.2	(-2%)
2000	40.5	40.7	(0%)	39.7	(-2%)	39.6	(-2%)	40.2	(-1%)
2500	43.3	43.9	(1%)	42.6	(-2%)	42.5	(-2%)	43.5	(0%)
5000	53.4	53.5	(0%)	51.8	(-3%)	52.4	(-2%)	54.4	(2%)
10000	61.5	61.6	(0%)	59.1	(-4%)	60.4	(-2%)	61.9	(1%)
20000	64.8	64.3	(-1%)	63.7	(-2%)	65.0	(0%)	65.5	(1%)

Table 6.3: Comparing factor storage with maximum observed memory used for delayed updates (`fox`, 8 threads, kilobytes).

n	Factor Memory	Simple	Max Child	Max Descendants	Fixed-CP	Weighted-CP
1000	3,911	864	720	1,775	2,028	3,211
2500	24,424	3,281	4,374	6,926	6,561	6,561
5000	97,676	9,032	9,935	12,644	11,741	12,644
10000	390,665	14,306	18,207	29,912	16,907	23,409

Table 6.2 shows the average results for the complete factorization of a random diagonally dominant matrix. We believe the averages to be reliable to within 0.2 Glop/s. They were obtained by running factorizations until we had accumulated at least 1 second worth of computation and then repeating this until the average was determined. However, we note that performance for methods Simple and Max Child had a much higher variation that other methods. We believe this to be due to the lack of tie breaking for update tasks and the consequent highly random execution sequence.

While it is disappointing that there is little difference between results, Weighted-CP seems to give the best results on large problems; however Max Child and Fixed-CP do better in the range $n = 1500$ to $n = 2500$. Following profiling for problems in this range we have determined that Max Child has many fewer updates that are delayed, resulting in less work than for Weighted-CP. This seems to be caused by a more random ordering of updates for Max Child — Weighted-CP gives all updates to the same block the same schedule, resulting in more collisions causing delayed updates. Fixed-CP does better than Weighted-CP on smaller problems, for which we offer the hypothesis that our choice of weighting is poor for small block sizes, where much of the time goes into the memory load rather than the floating-point operations. We note however that even on these small problems we saw evidence of Max Child causing task starvation mid-factorization due to a poor prioritisation of tasks. For larger problems the design assumptions of Weighted-CP are more accurate and the time spent in application of updates becomes negligible compared to other operations.

Table 6.3 shows the observed worst case memory usage for storage of our delayed updates. It confirms that in practice this does not become excessive. If we wished to minimise it, then we should choose either the Simple or Max Child schemes. As we are primarily concerned with speed for large matrices rather than memory usage, the comparisons in the remainder of this paper shall use the Weighted-CP schedule.

Table 6.4: Scalability of HSL_MP54 on a range of problem sizes using fox.

n	1 thread	2 threads		4 threads		8 threads	
	Gflop/s	Gflop/s	Speedup	Gflop/s	Speedup	Gflop/s	Speedup
500	5.6	8.6	1.5	13.4	2.4	17.7	3.2
1000	6.8	11.3	1.7	20.1	3.0	29.1	4.3
2500	7.6	14.5	1.9	26.9	3.5	43.5	5.7
5000	8.3	15.2	1.8	31.1	3.7	54.4	6.6
10000	8.6	17.1	2.0	33.6	3.9	61.9	7.2
20000	8.8	17.7	2.0	35.1	4.0	65.5	7.4

Table 6.5: Scalability of HSL_MP54 on a range of problem sizes using HPCx.

n	1 thread	2 threads		4 threads		8 threads		16 threads	
	Gflop/s	Gflop/s	Speedup	Gflop/s	Speedup	Gflop/s	Speedup	Gflop/s	Speedup
500	3.2	5.4	1.7	8.5	2.7	11.3	3.5	11.8	3.7
1000	4.0	7.2	1.8	12.9	3.2	19.6	4.9	25.7	6.4
2500	4.6	8.8	1.9	16.2	3.5	28.0	6.1	46.2	10.0
5000	4.8	9.4	2.0	18.1	3.8	32.0	6.7	53.4	11.1
10000	4.9	9.8	2.0	19.3	3.9	36.7	7.5	65.5	13.4
20000	4.9	-[1]	-	- [1]	-	39.3	8.0	69.9	14.2

[1] These results are not available due to time budget constraints

6.2 Scalability

Tables 6.4 and 6.5 demonstrate the scalability of HSL_MP54. We observe good scaling on large problems, however smaller problems do not scale quite so well. This is because the efficiency of the level 3 BLAS routines decreases with block size. In order to keep all threads fed for small problems on many threads a significantly smaller block size must be used than on a single thread, giving a lower efficiency than we would otherwise expect.

6.3 Block sizes

Table 6.6 shows the block sizes used on fox to achieve the results shown previously for the Weighted-CP schedule. We found that optimal block sizes are always multiples of 8, which we observe is the number of double precision numbers per cache line — this helps avoid false sharing.

The optimal block size seemed not to vary between the different schedulings, and was not very sensitive as long as nb was a multiple of 8, as is shown by Figure 6.3 for the case $n = 2000$.

Table 6.7 shows how the optimal nb varies with the number of threads in use. Based on these results we recommend that the user aims to have $nblk$ between 15 and 20 to estimate a near optimal value of nb for large problems.

6.4 Comparison with other codes

Figure 6.4 compares the performance of our code with the following codes:

HSL_MA54 Left looking Cholesky factorization code with fork-join parallelism of loops (version 1.3.0)

HSL_MP54 The DAG-based code described in this paper

MKL dpotrf Intel Math Kernel Library 10.0.1.014 full storage Cholesky factorization.

MKL dpptrf Intel Math Kernel Library 10.0.1.014 packed storage Cholesky factorization.

Table 6.6: Optimal block sizes used for Weighted-CP using 8 threads on `fox`.

n	nb	$nblk$
100	100	1
200	32	7
300	40	8
400	40	10
500	56	9
1000	104	10
1500	120	13
2000	184	12
2500	216	15
5000	340	15
10000	408	25
20000	968	21

Figure 6.3: Performance varying with nb for $n = 2000$ on 8 threads on `fox`.

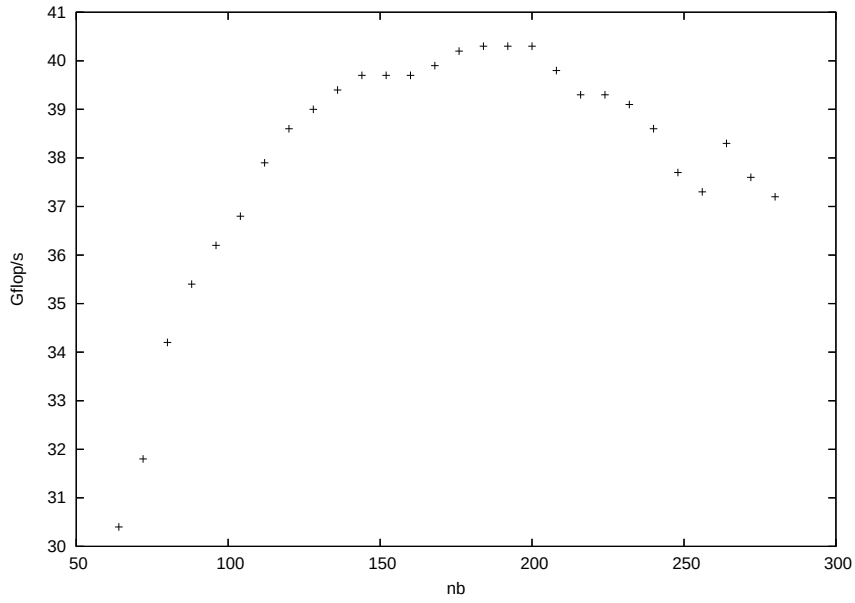
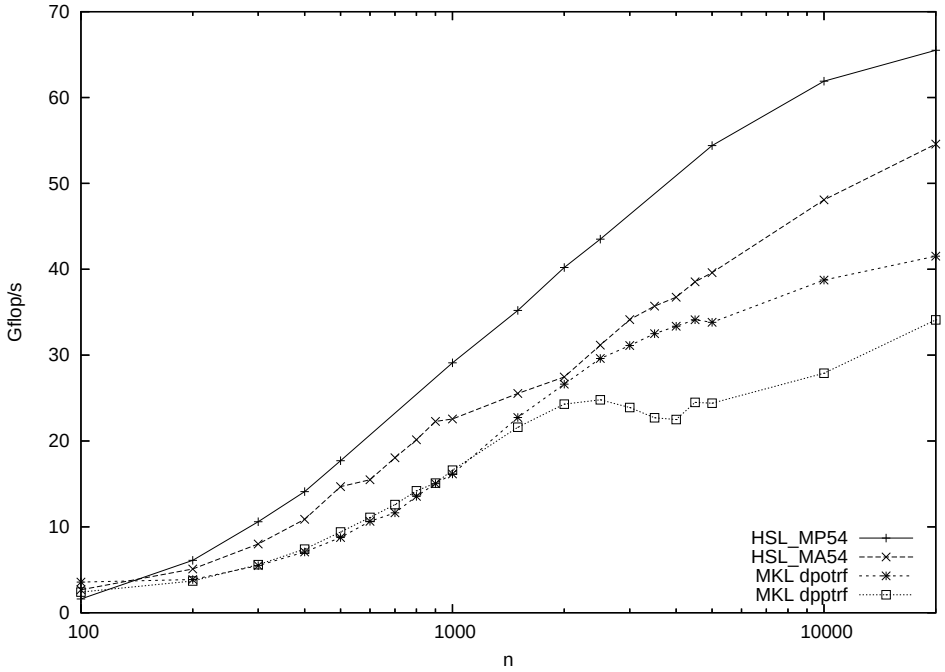


Table 6.7: Optimal block sizes for varying number of threads and n, with $nblk$ shown in brackets, using `fox`.

n	1 thread	2 threads	4 threads	8 threads
500	200 (3)	104 (5)	72 (7)	56 (9)
1000	200 (5)	156 (7)	128 (8)	104 (10)
2500	400 (7)	340 (8)	264 (10)	216 (12)
5000	400 (13)	480 (11)	400 (13)	340 (15)
10000	400 (25)	480 (21)	400 (25)	408 (25)

Figure 6.4: Performance varying with size of matrix n for complete factorizations, using 8 threads on `fox`.



Clearly for matrices that are not small `HSL_MP54` offers the best performance on 8 threads. For small matrices, for example $n = 100$, it is faster to factorize the matrix in serial due to caching issues and communication overheads.

Figure 6.5 shows a similar comparison on `HPCx`. Though we have not performed any tuning for this architecture beyond selecting good block sizes (which do not substantially differ from those on `fox`) we still get good performance, though the comparison with the vendor implementations of dpptrf and dpotrf are not quite so favourable except on small matrices.

7 Conclusions

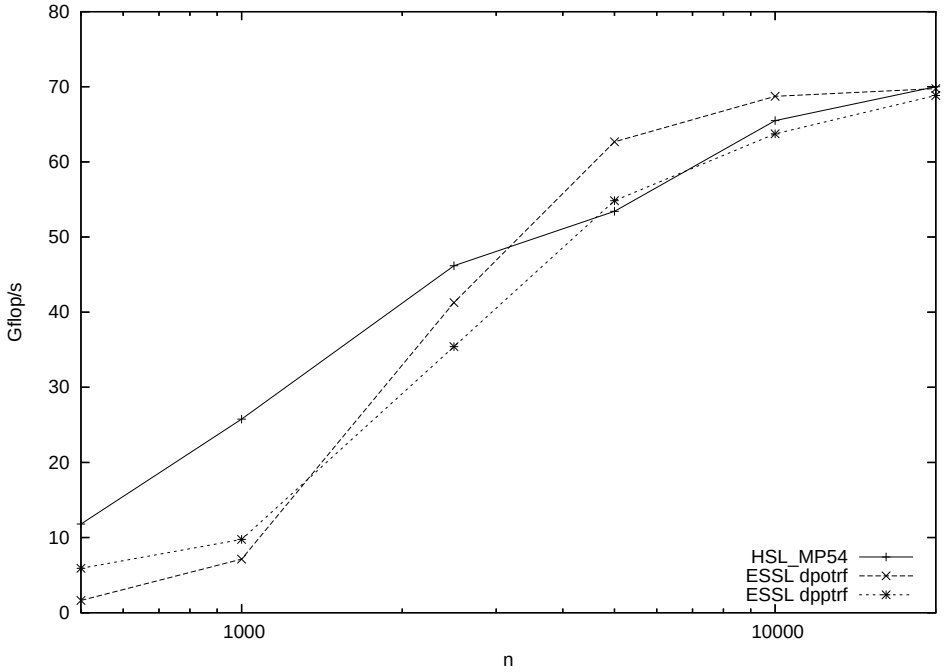
`HSL_MP54` is a Cholesky code that performs well on multicore machines, and should prove to be a good kernel for sparse multifrontal factorizations. However, care must be taken with small front sizes, possibly using a different factorization kernel, or exploiting parallelism at a higher level.

We have shown that the effect of different scheduling schemes in DAG-based Cholesky factorizations makes only a small difference, at most 7%. However, if the extra performance is considered worthwhile, a scheme that takes the critical path should be used. Further, the comparative times for different tasks must be taken into account when determining the critical path.

Future improvements to the scheduling may aim to reduce the number of threads attempting to update a single target block at once during update tasks, and may focus more on being cache efficient. The meaning of cache efficient may however change if all or many cores are sharing the same on-chip cache at some level.

It may also be of interest to replace our task handling mechanism with of OpenMP 3.0 tasks or Intel Thread Building Blocks.

Figure 6.5: Performance varying with size of matrix n for complete factorizations, 16 threads on `HPCx`.



8 Code availability

The code discussed in this paper has been developed for inclusion in the mathematical software library HSL. All use of HSL requires a licence. Individual HSL packages (together with their dependencies and accompanying documentation) are available without charge to individual academic users for their personal (non-commercial) research and for teaching; licences for other uses involve a fee. Details of the packages and how to obtain a licence plus conditions of use are available at `www.cse.clrc.ac.uk/nag/hsl/`.

9 Acknowledgements

We are grateful to our colleagues Iain Duff, John Reid and Jennifer Scott for useful discussions relating to this work. We also wish to thank Jack Dongarra and Alfredo Buttari for clarifications relating to their work.

References

[1] Anderson, Gunnels, Gustavson, J.K.Reid, and Wasniewski, *A fully portable high performance minimal storage hybrid format Cholesky algorithm*, ACM Transactions On Mathematical Software, 31 (2005), pp. 201–227.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, third ed., 1999.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[4] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, *The impact of multicore on math software*, in Proceedings of Workshop on State-of-the-art in Scientific and Parallel Computing (Para06), 2006.

[5] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Tech. Rep. UT-CS-07-600, ICL, 2007. also LAPACK Working Note 191.

[6] HSL, *A collection of Fortran codes for large-scale scientific computation*, 2007. See http://www.cse.clrc.ac.uk/nag/hsl/.

[7] F. G. V. Zee, P. Bientinesi, T. M. Low, and R. A. van de Geijn, *Scalable parallelization of flame code via the workqueuing model*, ACM Transactions On Mathematical Software, 34 (2008).